Novel Beetle Algorithm for Cartesian Grid Generation in Two Dimensions

Ashok Srivastava* and K. S. Ravichandran†
National Aerospace Laboratories, Bangalore 560 017, India

Introduction

CARTESIAN grid methods find applications in flow computations around complex geometry. The task is accomplished in two major steps, namely, grid generation by employing a suitable algorithm and flow computations using an appropriate scheme. This Note gives a description of an innovative Beetle algorithm, developed by the authors, which generates a Cartesian grid around arbitrary shapes in two dimensions. Different approaches have been used by researchers in devising methods for Cartesian grid generation around complex shapes, and the reader may refer to Refs. 1–3 for details about Cartesian mesh-generation methods.

A robust Cartesian grid generator has been developed for arbitrary geometry in two dimensions, with built-in features of automation in grid generation. The rectangular cells, which are located at the fluid-boundary interface in the form of Cut_Cells, are collated separately from the external UnCut_Cells in the Cartesian mesh. The interior cells within the boundary, where the flow does not exist, are separately assembled by a sorting algorithm. A specified arbitrary curve is immersed in a Cartesian mesh with parametric specifications of grid spacing in the two orthogonal directions (x, y). A capturing technique has been used to pick up intersections of the specified geometry curve and the Cartesian cell walls. Figuratively, an intelligent Beetle moves along the prescribed geometry curve, which senses an approaching cell wall by a user-defined capture distance. After capture the Beetle slows its movement (finer step size) until the error between its location and the cell wall reaches a predefined minimum. The Beetle traverses the entire curve representation of the given configuration in step sizes specified by the user. Therefore, it is impossible to miss any intersection point in the Cartesian mesh within the specified resolution. This also ensures availability of the curved path traced by the Beetle within a cell. Multiple intersections within the same cell are also captured. The specified geometry curve can also intersect itself several times. The Beetle algorithm offers a versatile Cartesian grid generation around arbitrary shapes in two dimensions.

Grid Definitions and Data Integrity

The configuration input geometry is defined as a string of tabulated coordinates (x, y) of the specified boundary curve. The basic Cartesian mesh, in which the specified geometry is immersed, is defined by the number of cells in the x and y directions (nx, ny). A rectangular box enclosing the geometry is computed. The computational flow domain is then defined by extending the Cartesian mesh to specified distances from the edges of this bounded rectangle, which is further discretized into rectangular cells. These cells are generated by I lines and J lines, which run parallel to the (x, y) axes of the reference system.

The Cartesian mesh is categorized into three distinct domains comprising the virgin Un_Cut_Cells, the boundary-intercepted Cut_Cells, and the untouched Internal_Cells. The grid-generation code collects the intercepted cells separately, and, therefore, the flow domain can be considered, either, in the external region from the boundary, or alternatively, in the interior of the specified geometry for internal flow computations.

Data integrity can be ensured at two levels. The user specification of the data is presumed to ensure an accurate representation of the configuration and is the first level of data integrity. At the second level the data are modified appropriately by interpolation techniques to generate additional points in the computational domain. A higher-order interpolation could be used to generate such additional points to a desired accuracy. The end point of the specified geometry coincides with its starting point. This fact is used as a stopping condition for the Beetle in its onward march along the boundary.

Advancing Beetle

A Beetle (represented by a moving point) commences its journey from the starting position of the specified geometry in its world of the Cartesian mesh. A subroutine (BUGP) gets the initial position of the Beetle and checks if it is already located on an edge of a Cartesian cell. As the Beetle moves, the coinciding of its location with the Cartesian cell walls is collected as a string of intercepted points. From the initial location, or from an intercepted point, the probable directions in which the Beetle can move are evaluated by a subroutine (BUGD), which identifies the indices of the Cartesian cell edges (N, S, E, W) toward which it can advance. The Beetle can be located 1) on a corner of a cell, 2) on an I line, 3) on a J line, or 4) at a point in the interior of a cell. As the Beetle moves, it can intercept another corner, an I line, or a J line. The required conditions for such intersections are outlined next:

1) The Beetle moves from an intersected point 1(K) to the next intersected point 2(K + 1).

2) Checks made on point 1(K).

3) If the Beetle is located on an I line, EDGEI(K) = 1, and EDGEJ(K) = 0.

4) If the Beetle is located on a J line, EDGEI(K) = 0, and EDGEJ(K) = 1.

5) If the Beetle is located on a corner, EDGEI(K) = 1, and EDGEJ(K) = 1.

The following conditions are applicable, depending on how the Beetle traverses in the Cartesian mesh to a second point 2(K + 1):

1) The Beetle moves from a corner point (point 1) to its next interception point (point 2) (Fig. 1a). Twenty cases have been identified for possible Beetle movements in this subcategory. A detailed description is available in Ref. 4. For example, when the Beetle moves from a corner point (point 1) and intercepts the west wall (case 1), the required conditions are the following: IB(K) = IB(K + 1) − 1; JB(K) = JB(K + 1); EDGEI(K)=EDGEI(K+1); and EDGEJ(K)= EDGEJ(K+ 1)−1.

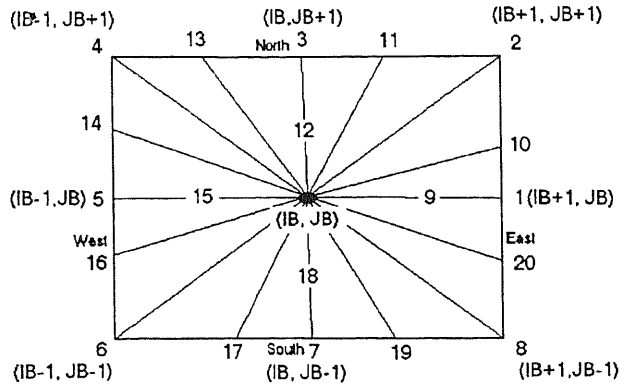
2) The Beetle moves from an I line (point 1) to its next interception point (point 2) (Fig. 1b). Twelve cases have been identified for possible Beetle movements in this subcategory. For example, when the Beetle moves from an I line (point 1) and intercepts the north wall (case 4), the required conditions are the following: IB(K) = IB(K + 1); JB(K) = JB(K + 1) − 1; EDGEI(K) = EDGEI(K + 1) − 1; and EDGEJ(K + 1) = 0.

3) The Beetle moves from a J line (point 1) to its next interception point (point 2). This category is similar to category (B) with the indices (I, J) interchanged, and similar conditions can be envisaged.

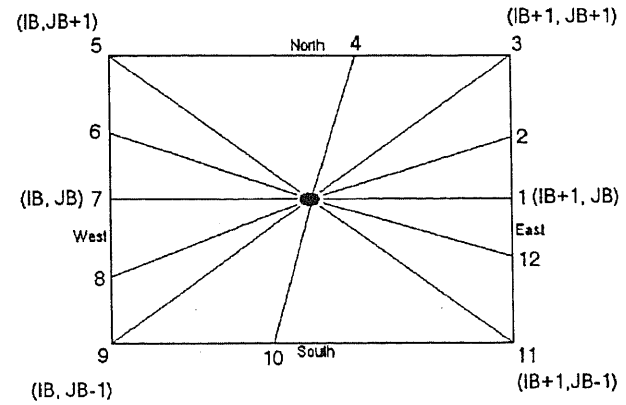
A resolution factor (RSLN) is defined as a fraction of the length/breadth of the Uncut-Cell (say, RSLN = 0.01), and the subsequently discussed Beetle movement parameters are defined in terms of RSLN. The Beetle starts moving on the specified curve in moderate steps (step-size DSRUF), keeping track of its distance from all of the four sides of the cell surrounding it (N, S, E, W). These distances (CHKN, CHKS, CHKE, CHKW) are continuously compared with a parametric capture distance (ERRC). If the next interception is on an I line or on a J line, one of these checked distances falls below the capture value. If the Beetle is approaching a corner point, then two of these checked distances fall below the capture distance. The step size of the Beetle's onward march is considerably reduced after capture (DSFIN). The intersection is obtained when the error is less than the specified error (ERR). Subsequently, the conditions

Received 14 September 1999; revision received 18 April 2000; accepted for publication 28 April 2000.

*Scientist, Computational and Theoretical Fluid Dynamics Division.
†Assistant Director, Computational and Theoretical Fluid Dynamics Division.



a) Beetle starts from a corner of a cell (IB, JB)



b) Beetle starts from an I-line of a cell (IB, JB)

Fig. 1 Beetle traverse on a Cartesian cell.

outlined in the 1), 2), and 3) determine the location of the Beetle after interception. The Beetle is advanced a little ahead of the intersection point so that the interception is fully computed, and, then, it is positioned back to the interception point for its further movement. This ensures that a corner point is picked up as an intersection, which otherwise could be mistaken to be simply an I line or a J line intersection.

The Beetle may GRAZE along the Cartesian mesh walls depending on how the boundary of the configuration geometry is specified. A GRAZING test is conducted after each interception and is accounted for in the movement of the Beetle. After grazing a cell wall, the Beetle can move to a corner point of the same cell, or it can leave the cell wall after partial grazing. It can also graze several cell walls. It can also suddenly commence grazing a J line after grazing an I line. All such possibilities have been appropriately identified and implemented in the code.

Assembly of Cut-Cell Vertices

After the intersection points have been collated, a subroutine (CELLP) is used to construct the Cut_Cells from the rectangular cells. These Cut_Cells are derived by the inclusion of the intercepted points as the cell vertices. As the Beetle moves ahead in "strides" along the intercepted points, it divides a rectangular Cartesian cell into a left Cut_Cell and a right Cut_Cell, with reference to its advancing direction (anticlockwise). The Cut_Cell construction is then performed by adjusting the cell corners that are appropriately modified by inclusion of the intersection points, which are added as vertices. The details of Cut_Cell construction are given in Ref. 4.

A check is continuously made for the traversing Beetle making multiple intersections within a cell, indicating its confinement in a particular cell. Any three consecutive intersection points (P1, P2, P3) are tested for their location in the same cell. The two line segments P1-P2 and P2-P3 must satisfy certain conditions to be in the same cell. Details are available in Ref. 4. Usually, such multiple intersections can be avoided by refining the basic Cartesian mesh or by introducing adaptive mesh refinement locally.

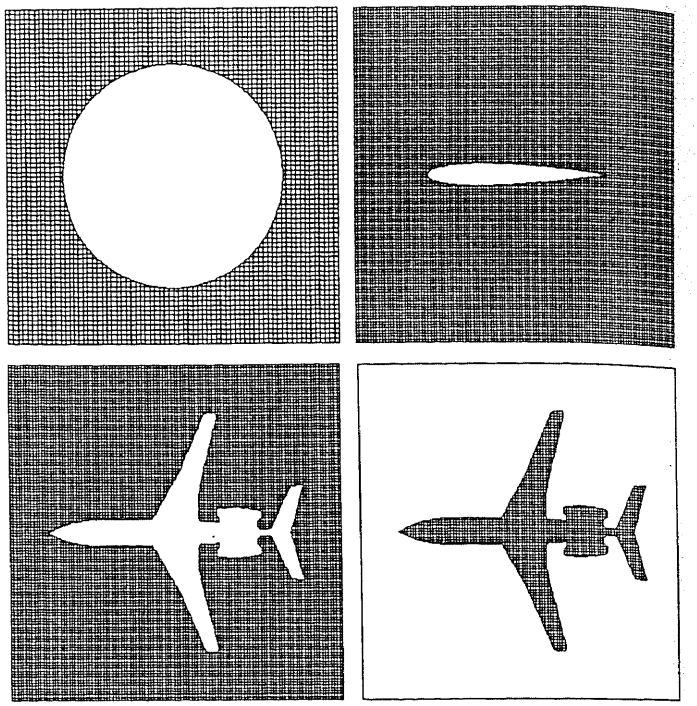


Fig. 2 Cartesian grid for interior and exterior domains.

Separating Inside/Outside Regions

The two domains of interest in flow computations are the external and the internal grid within, or outside of, the prescribed geometry. The separation of inside/outside regions is carried out by a subroutine (MAP_IO). The user specifies a point located inside the boundary curve as data input. All of the cells in the Cartesian mesh are initially switched on with a property NBUGC = 1. The algorithm then proceeds to check the neighboring cells recursively, which are switched off by redefining the property NBUGC = 0, until a boundary cell (Cut_Cell) is encountered, which has a property NBUGC = 2. This procedure separates the inside region from the outside, systematically.

A Cartesian grid visualizing postprocessor[5] developed in C++ is then used for graphical viewing of the generated Cartesian grid.

Beetle Tracking Parameters

The parameters employed in the Beetle algorithm are sensitive to how mathematical zero is defined in the machine and the resolution RSLN, discussed earlier. The machine zero can be defined as ZERO = 1.0E-06 as parametric data input within the grid-generation code. The parameter RSLN is a fraction of an Un-Cut cell side length. The other parameters are defined as per the accuracy demand of the user for the generation of the Cut_Cells. A typical set of these parameters is given next:

$$\text{ZERO} = 1.0\text{E-}06; \qquad \text{RSLN} = 0.01$$

$$\text{DSRUF} = \text{RSLN}; \qquad \text{DSFIN} = 0.1 \times \text{RSLN}$$

$$\text{ERRC} = 2.0 \times \text{DSRUF}; \qquad \text{ERR} = 1.0 \times \text{DSFIN}$$

Sample Cases

The following sample cases demonstrate the application of the grid-generation code based on the Beetle algorithm (Fig. 2): 1) circle, 2) airfoil, and 3) aircraft.

The Cartesian visualizing code[5] has been used to create the graphic images. The versatility of the grid generation for arbitrary shapes is clearly evident in the samples (1–3). Figure 2 also depicts internal cells within the aircraft (computations for air conditioning).

Conclusions

A Cartesian grid generator has been coded to obtain Cartesian grid around arbitrary geometry in two dimensions. The grid generation is accomplished by a novel technique employing a Beetle algorithm.

The significant parameters in defining the movement of the Beetle and its capture of intersection points have been ascertained and implemented within the code. The sample grid generated around several shapes shows the versatility of the code. It is intended to extend this methodology to shapes in three dimensions by converting the movement of the Beetle to that of a "paint brush" (defined by an elemental arc), which traverses an arbitrary surface (paints the surface), and thereby collating the intersecting boundaries of the hexahedral cells. A flow solver based on Colella et al.[1] is being developed to generate flow solutions.

References

[1]Pember, R. B., Bell, J. B., Coelella, P., Crutchfield, W. Y., and Welcome, M. L., "An Adaptive Cartesian Grid Method for Unsteady Compressible Flow in Irregular Regions," *Journal of Computational Physics*, Vol. 120, No. 2, 1995, pp. 278–304.

[2]Quirk, J. James, "A Cartesian Grid Approach with Hierarchical Refinement for Compressible Flows," Inst. for Computer Applications in Science and Engineering Rept. 94-51, NASA CR 194938, June 1994.

[3]Zeeuw, Darren De, and Powell, Kenneth G., "An Adaptively Refined Cartesian Mesh Solver for the Euler Equations," *Journal of Computational Physics*, Vol. 104, No. 1, 1993, pp. 56–68.

[4]Srivastava, A., and Ravichandran, K. S., "Cartesian-Grid Generation in Two Dimensions," National Aerospace Lab., NAL-PD-CF 9903, Bangalore, India, Feb. 1999.

[5]Srivastava, A., and Ravichandran, K. S., "Cartesian-Grid-Visualization Code (C++)," National Aerospace Lab., NAL-PD-CF 9906, Bangalore, India, July 1999.

Comparison of Deterministic and Stochastic Optimization Algorithms for Generic Wing Design Problems

X. Wang* and M. Damodaran†
Center for Advanced Numerical Engineering Simulations, Nanyang Technological University, Singapore 639798

Introduction

IN the past decade, a variety of numerical optimization algorithms have been extensively used to address multidisciplinary design optimization (MDO) problems that deal with design processes that are dependent on interactions among several engineering disciplines. MDO consists of many challenging features, such as a heterogeneous mix of analysis codes for evaluating objective functions, a large number of design variables, discrete design parameter values, and complex constraints. Local optimization strategies, such as gradient-based algorithms, have been widely applied to engineering system designs.[1,2] Although the number of design iterations required by local optimizers can be small, a major shortcoming of local optimizers is that they may get trapped in local optima. Although the performance of local optimizers can be enhanced by sensitivity analysis, such as that given in Ref. 2, which reduces the number of objective function evaluations in the optimization process, it is often accomplished at the expense of additional work in constructing models for estimating sensitivities. In some cases the use of approximate derivatives in estimating sensitivities (see Ref. 1) can lead to a loss of accuracy in the optimization process.

Received 10 November 1999; revision received 5 January 2000; accepted for publication 20 March 2000.

*Research Fellow. Member AIAA.
†Associate Professor, School of Mechanical and Production Engineering; mdamodaran@ntu.edu.sg. Senior Member AIAA.